VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Absolvování individuální odborné praxe

# Individual Professional Practice in the Company

2020 Marcel Ševec

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Zadání bakalářské práce

Student: **Marcel Ševec**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Absolvování individuální odborné praxe
Individual Professional Practice in the Company

Jazyk vypracování: slovenština

Zásady pro vypracování:

1. Student vykoná individuální praxi ve firmě: EAGO SYSTEMS s.r.o.
2. Struktura závěrečné zprávy:
   a) Popis odborného zaměření firmy, u které student vykonal odbornou praxi a popis pracovního zařazení studenta.
   b) Seznam úkolů zadaných studentovi v průběhu odborné praxe s vyjádřením jejich časové náročnosti.
   c) Zvolený postup řešení zadaných úkolů.
   d) Teoretické a praktické znalosti a dovednosti získané v průběhu studia uplatněné studentem v průběhu odborné praxe.
   e) Znalosti či dovednosti scházející studentovi v průběhu odborné praxe.
   f) Dosažené výsledky v průběhu odborné praxe a její celkové zhodnocení.

Seznam doporučené odborné literatury:

Podle pokynů konzultanta, který vede odbornou praxi studenta.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **doc. Mgr. Jiří Dvorský, Ph.D.**

Konzultant bakalářské práce: Ing. Aleš Daněk

Datum zadání: 01.09.2019
Datum odevzdání: 30.04.2020

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA Fakulta elektrotechniky a informatiky 1D16 OSTRAVA

doc. Ing. Jan Platoš, Ph.D.
*vedoucí katedry*

prof. Ing. Pavel Brandštetter, CSc.
*děkan fakulty*

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

V Ostrave 20. apríla 2020

.....

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v bakalářských studijních programech VŠB-TU Ostrava

V Ostrava 23. dubna 2020


.....

Rád by som chcel poďakovať vedeniu firmy Eago systems spol. s.r.o., že mi dali možnosť u nich aboslovovať individuálnu odbornú prax. Zároveň by som chcel poďakovať môjmu konzultantovi, pánovi Ing. Alešovi Daněkovi, ktorý si vždy našiel na mňa čas, keď som potreboval konzultovať riešenie mojej práce. A nakoniec by som chcel poďakovať môjmu vedúcemu práce, pánovi doc. Mgr. Jiřímu Dvorskému, Ph.D. za jeho konzultácie pri vytváraní bakalárskej práce.

**Abstrakt**

Táto bakalárska práca popisuje priebeh absolvovanie individuálnej odbornej praxe v spoločnosti EAGO, s.r.o. Obsahuje informácie o firme, kontkrentnom mojom zadaní, technológiách, ktoré som potreboval použiť a o celkovom riešení projektu.

**Kľúčové slová**: Java, Maven, Netbeans, Wildfly, Kafka, Wicket

**Abstract**

This bachelor thesis describes the process of completion of individual professional practice in the company EAGO, s.r.o. It contains information about the company, my assignment, the technologies which I needed to use and the solution of the project.

**Keywords**: Java, Maven, Netbeans, Wildfly, Kafka, Wicket

# Obsah

# Zoznam použitých skratiek a symbolov

| | | |
|---|---|---|
| ID | – | Identifikačné číslo |
| JDK | – | Java Development Kit |
| JVM | – | Java Virtual Machine |
| RDS | – | Radio Data System |
| TMC | – | Traffic Message Chanel |
| SDR | – | Software Defined Radio |

# Zoznam obrázkov

# Zoznam výpisov zdrojového kódu

# 1 Úvod

Ako bakalársku prácu som sa rozhodol, vybrať možnosť absolvovania individuálnej odbornej praxe v spoločnosti Eago systems spol. s.r.o.. Využil som túto možnosť z viacerých dôvodov. Po prvé som si chcel obohatiť svoje teoretické znalosti zo školy o nové, ktoré sa využívajú v praxi. Aj keď naša škola nám dala veľa znalosti o technológiách, ktoré sa vo svete momentálne používajú stále, je ešte veľa veci a technológií, ktoré si musíme doštudovať sami a táto firma, mi niektoré ukázala a dala možnosť sa ich naučiť.

Ďalším dôvodom prečo som sa rozhodol pre túto možnosť je zistiť, ako fungujú programátorské firmy a takisto zistiť, ako sa využívajú technológie, ktoré som poznal len teoreticky aj v praxi. Posledným dôvodom bola možnosť nazbierať cennú a dôležitú prax vo firme, ktorú budem určite potrebovať pri hľadaní nových pracovných zamestnaní, nakoľko prax v dnešnej dobe je veľmi dôležitá. Táto firma pre mňa nie je niečo neznáme, keďže pracujem v nej už dva roky a prešiel som si viacerými pozíciami od testera až po programátora, ktorým som do dnes.

V tomto dokumente popisujem moju prácu na projekte, ktorý mi bol zadaný vedúcim a následne schválený garantom. Celý dokument sa skladá z viacerých časti. V prvej časti sa nachádzajúcu informácie o firme a takisto tu popisujem akým projektom sa táto firma venuje. Ďalšia časť obsahuje zoznam použitých technológií, kde som sa ku každej snažil v stručnosti popísať, ako funguje, či načo sa používa. Treťou časťou je zadanie, ktoré mi bolo dané firmou, nachádza sa tu v stručnosti popísané, ako celý projekt má fungovať.

Predposlednou časťou je celkové riešenie projektu. V tejto časti popisujem jednotlivé kroky, pri vykonávaní projektu. V daných krokoch uvádzam, aké technológie či knižnice používam. Opisujem tu aj problémy, s ktorými som sa tu stretával a ako som sa ich snažil vyriešiť. Poslednou časťou je záver, kde zhodnocujem celý priebeh práce na projekte.

# 2 O firme

Spoločnosť Eago systems spol. s.r.o. [1] bola založená Zdeňkem Markem a Petrem Pivodou 19. augusta 1994, predtým známa pod názvom Eastwood Bohemia spol. s.r.o. Táto spoločnosť začala najskôr s poskytovaním konzultácií v oblastiach manažmentu, finančného riadenia a ďalších.

Neskôr, ale začala poskytovať podporu v IT oblasti. Firma vlastní vlastné technické prostriedky a snaží sa byť na vrchole, či už v kyberbezpečnosti, alebo GDPR. Kladie veľký dôraz na najnovšie a najpokročilejšie technológie a na kvalitných zamestnancov. Momentálne sa venuje viacerým projektom či už pre českých, ale aj zahraničných zákazníkov. Najväčšie projekty sú:

- Systém pre monitorovanie výbušnín, munície a zbraní. Tento projekt bol vytvorený pre políciu českej republiky a používa sa dodnes.
- Systém pre poisťovne, pomocou ktorého sa efektívne organizujú výjazdy zásahových vozidiel. S týmto projektom firma stále napreduje a nie len v Čechách, ale aj v iných krajinách v rámci Európy.
- Systém pre manažment dopravných obmedzení.
- Systém pre vysielanie dopravných informácií.

V tejto spoločnosti pracujem na pozícii junior programátor a venujem sa hlavne programovaniu v jazyku Java, ale aj spravovaniu databázových systémoch. Aplikácia, ktorú som v priebehu absolvovania individuálnej odbornej praxe vytváral, bude využitá v novom projekte pre nášho zákazníka.

# 3 Použité technológie

## 3.1 Java

Java [2] je objektovo orientovaný programovací jazyk, ktorý bol predstavený v roku 1995 spoločnosťou Sun Microsystem. Tento programovací jazyk je vyvíjaný ako open source a patrí medzi najpoužívanejšie programovacie jazyky na svete.

Pre vývoj programu v jazyku Java je potrebná sada Java Development Kit (JDK). Zdrojový kód v Jave je narozdiel od ostatných programovacích jazykoch, ktoré kompilujú kód do strojového kódu, skompilovaný do bajtového kódu a ten sa nasledovne ukladá do súboru s príponou .class.

Hlavnou výhodou programovacieho jazyka Java je, že je multiplatformový, čiže môžeme s ním pracovať na viacerých platformách a strojoch, ktoré majú nainštalovaný JVM (Java Virtual Machine). JVM vykonáva postupne inštrukcie, ktoré sú skompilované bajtovom kóde.

## 3.2 Netbeans

Netbeans [3] bol študentský projekt na Karlově univerzitě, ktorý odkúpila spoločnosť Sun MicroSystem a v jeho vývoji pokračuje Oracle. Netbeans je voľne šíriteľné vývojové prostredie.

Dokáže pracovať na rôznych platformách, ako je Windows, Linux, či Mac OS. Pre prácu ponúka viacero balíčkov, ako je balíček pre C/C++, JavaScript a ďalšie. Hlavným balíčkom je ale Java. Ďalej obsahuje viaceré nástroje či už pre vývoj webových, alebo aj mobilných aplikácií. Takisto poskytuje podporu Maven.

## 3.3 Maven

Maven [4] je nástroj, ktorý bol vytvorený predovšetkým pre programovací jazyk Java. Hlavnou úlohou Maven je zjednotené a zjednodušené zostavovanie projektov.

Pomocou Maven vieme ľahko spravovať závislosti na iné projekty a knižnice pomocou pridaného súboru pom.xml. Maven automatický pridá ku projektu všetky knižnice, ktoré sú vypísané v danom súbore pom.xml. Súbor pom.xml takisto obsahuje všetky potrebné informácie o danom projekte, ako napríklad názov, verzia atď. Príklad súbora pom.xml je možné vidieť na výpise 1.

Tento súbor musí, ale minimálne obsahovať group-id, artifact-id a version.

```
<project>
<modelVersion>4.0.0</modelVersion>
<groupId>cz.eago</groupId>
<artifactId>rdsdata</artifactId>
<version>1.0-SNAPSHOT</version>
</project>
```

Výpis 1: Príklad súbora pom.xml

## 3.4 PostgreSQL

PostgreSQL [5] je objektovo relačný databázový systém, ktorý je voľne šíriteľný a dostupný na viacerých platformách, vrátane operačných systémoch Windows, Mac OS X a Linux.

Patrí medzi najpoužívanejšie databázové systémy a to hlavne vďaka svojej bezpečnosti a spoľahlivosti. Spĺňa všetky požiadavky ACID (Atomičnosť, Konzistencia, Izolácia, Robustnosť). PostgreSQL používa aj najnovšie dátové typy, ako sú JSON a XML.

PostgreSQL nám takisto umožňuje vytvárať procedúry a triggery. Jeho výhodou je aj rozšíriteľnosť o vlastné dátové typy a nové doplnky, ako je napríklad PostGIS. PostGIS bol vytvorený pre podporu geografických informačných systémoch.

Programovacie jazyky ako Java, obsahujú rôzne knižnice pre prácu s týmto databázovým systémom.

## 3.5 Wildfly

Wildfly [6] je aplikačný server, vyvinutý firmou RedHat. Je napísaný v jazyku Java a dostupný na viacerých platformách, ako je Windows, Mac OS X, či Linux. Predtým známy pod názvom Jboss. Wildfly poskytuje najnovšie štandardy pre vývoj webových aplikácií. Vďaka podpore Web socket umožňuje optimálnu a plne duplexnú komunikáciu medzi aplikáciami. Wildfly architektúra nám umožňuje, podľa potreby pridávať a odoberať jednotlivé subsystémy.

## 3.6 Apache Wicket

Apache Wicket [7] je framework určený pre vytváranie webových aplikácií pomocou jazyka Java. Vďaka Wicketu môžeme ľahšie vytvárať webové aplikácie.

Vo Wickete sa každá stránka, či panel skladá z dvoch súboroch. Jeden súbor je .html, ktorý je určený pre vytváranie statického obsahu stránky. Druhý súbor je .java, ktorý je určený pre dynamickosť a funkcionalitu jednotlivých časti webu. Aby Java kód vedel, s ktorou časťou webovej stránky ma pracovať, je potrebné v html kóde pridávať atribúty wicket:id k jednotlivým tágom.

## 3.7 Apache Kafka

Kafka [8] je voľne šíriteľná softvérová streamovacia platforma, ktorá bola vytvorená v jazyku Java a Scale. Kafka slúži na posielanie správ medzi jednotlivými aplikáciami. Jej hlavnou výhodou je rýchlosť a odolnosť voči chybám.

Táto streamovacia platforma beží ako cluster na jednom, alebo viacerých serverov a ukladá jednotlivé správy podľa názvu tém zvaných topics. Čiže jednotlivé správy sa posielajú v rámci topicov. Každá správa sa skladá z kľúča, hodnoty a časového razítka. Kafka sa skladá zo štyroch častí.

- Producer API– umožňuje aplikáciám posielať správy do jedného, alebo viacerých topicov.

- Consumer API– umožňuje aplikáciám spracovávať správy z jedného, alebo viacerých topicov.
- Stream API– umožňuje aplikáciám spracovávať správy z jedného, alebo viac topicov a zároveň posielať správy do jedného, alebo viac topicov.
- Connector API– umožňuje vytváranie a spúšťanie producerov či consumerov, ktoré sú pripojené k jednotlivým topicom existujúcich aplikácii.

## 3.8 Rest

Rest [9] je architektúra rozhrania pre distribúciu dát. Rest bol vytvorený pre ľahký a jednotný prístup k zdrojom. Predstavuje vlastne prostredie, s ktorým môžeme vykonávať CRUD operácie. Pomocou restu môžeme pracovať s rôznymi dátami, ale najpoužívanejšie sú JSON a XML. Každý zdroj musí mať vlastnú URI. Príklad URI: `scheme://host:port/path?query#fragment`. Rest ma 4 základne metódy.

- Get: slúži k získavaniu údajov, alebo dát. Klient posiela požiadavku na potrebné dáta a server posiela odpoveď obsahujúcu dané dáta.
- Post: slúži na vytváranie nových dát. Klient posiela dáta na server a server by mal vrátiť kód 201.
- Delete: slúži na vymazávanie dát.
- Put/Update: slúži na aktualizáciu dát.

# 4 Zadanie práce

Úlohou študenta bude implementovať systém pre zhromažďovanie a vizualizáciu TMC dát z vysielania FM rádií. Napríklad v Českej republike vysielajú, iba rádia Vltava a Čas rádio.

Samostatný zber dát z éteru je už vyriešený a bude predávaný na jeho vstupný modul formou JSON token streamu s informáciami popísaných formou Alert-c a väzbou na lokačné tabuľky. Samotný systém sa bude skladať z nasledujúcich moduloch.

- modul-1: web modul s rozhraním REST
- modul-2: modul pre spracovanie dát a ich doplnenie o GPS dáta
- modul-3: vizualizačný modul/web s podporou zobrazenia informácií v mape

Jednotlivé moduly na seba priamo nevidia a sú prepojené pomocou streamovacej platformy Apache Kafka. Dáta ukladá modul-2 do NoSQL databáze Apache Cassandra, modul-3 z tejto databáze, iba číta. Technologický je riešenie postavené ako niekoľko war aplikácií bežiacich pod aplikačným serverom Wildfly na Linuxe.

Použitý programovací jazyk je primárne Java, v prípade vizualizácie i JavaScript a framework Apache Wicket. Projekty sú zostavované nástrojom Apache Maven. Študent sa v rámci implementácie naučí spomínané technológie používať a konfigurovať v miere potrebnej pre sprovoznenie celého systému.

Splnenie celého zadania mi trvalo približne päťdesiatpäť dní v prepočte štyristoštyridsať hodín a to aj s preštudovaním zadania a všetkých technológii, ktoré som potreboval využiť a celkovou implementáciou vrátanie testovania. Najkratšie mi trvalo implementovať modul-1 a naopak, najdlhšie mi zas trvalo vytvoriť modul-3.

# 5 Riešenie

Prvé týždne som sa zoznamoval s daným zadaním a konzultoval som ho s mojim konzultantom vo firme. Firma mi pripravila zadanie s menej konkrétnymi požiadavkami a celkové riešenie nechala na mne.

Na implementáciu sme sa dohodli, že nebudem používať žiadne interné knihovne. Následne som dostal vzorku vstupných dát do modulu-1 a dohodli sme sa, že všetko budem robiť pod virtuálnym strojom. Počas ďalších dní som nainštaloval VirtualBox, operačný systém Linux a nainštaloval Netbeans a JDK. Stiahol a nakonfiguroval streamovaciu aplikáciu Apache Kafka a nakoniec nainštaloval a nakonfiguroval databázu PostgreSQL, ktorej som pridal doplnok PostGIS.

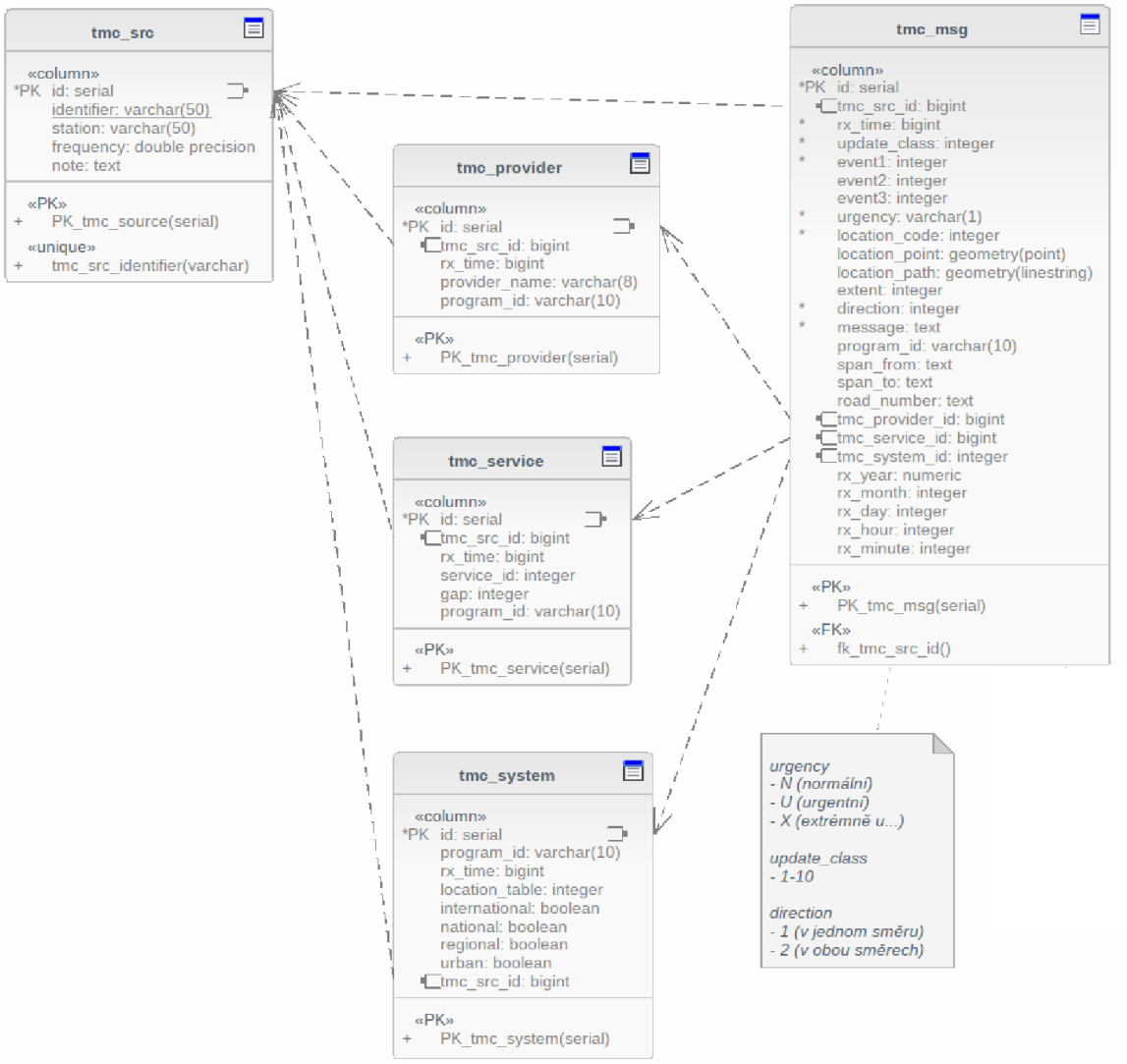
## 5.1 Databáza

Na začiatku som mal zadanie, aby som pracoval s databázou Cassandra, ale po konzultácií s vedúcim sme sa dohodli pre databázu PostgreSQL, keďže Cassandra je databáza skôr na veľké dáta, ktoré som v danom zadaní nepoužíval. Celkovo Cassandra je zložitejšia na používanie a nepodporuje všetky SQL funkcie. V Cassandre pri vytváraní tabuliek si tiež treba dobre rozmyslieť rozloženie jednotlivých stĺpcov.

Pri rozhodovaní, ktorú databázu použiť ako náhradu namiesto databázy Cassandra sme brali do úvahy niekoľko veci. Chceli sme databázu, ktorá je užívateľský prijateľnejšia, je open source a podporuje väčšinu SQL funkcií. Taktiež sme potrebovali databázu, ktorá bude podporovať geografické dátové typy. Preto sme sa rozhodli pre databázu PostgreSQL, ktorá tieto požiadavky spĺňala.

Po nainštalovaní databázy, som vytvoril databázovú štruktúru s PostGISom. Celú databázovú štruktúru som prekonzultoval s vedúcim, aby vyhovovala aj do prípadných budúcich projektov Štruktúra sa skladá z piatich tabuliek. Celú štruktúru je možné vidieť na obrázku 1.

- Tabuľka `tmcsrc` obsahuje stĺpce reprezentujúce záznamy z identifikačného tokenu a obsahuje aj stĺpec id, ktorý je primárnym kľúčom danej tabuľky.
- Tabuľka `tmcprovider` obsahuje stĺpce reprezentujúce konfiguračný token 1. Obsahuje ale aj stĺpec id, ktorý je primárnym kľúčom a stĺpec `tmc_src_id` reprezentujúci cudzí kľúč na tabuľku `tmcsrc`.
- Tabuľka `tmcservice` obsahuje stĺpce reprezentujúce konfiguračný token 2 a stĺpce id– primárny kľúč a `tmcs_src_id`– cudzí kľúč na tabuľku `tmcsrc`.
- Tabuľka `tmcsystem` obsahuje stĺpce reprezentujúce konfiguračný token 3 a stĺpce id– primárny kľúč a `tmcs_src_id`– cudzí kľúč na tabuľku `tmcsrc`.




Obr. 1: Grafické znázornenie databázovej štruktúry

- Tabuľka `tmcmessage` obsahuje stĺpce reprezentujúce vlastný token a stĺpce id– primárny kľúč, `tmc_src_id`– cudzí kľúč na tabuľku `tmcsrc`, `tmc_provider_id`– cudzí kľúč na tabuľku `tmcprovider`, `tmc_system_id`– cudzí kľúč na tabuľku `tmcsystem`.

## 5.2 TMC JSON tokeny

Na začiatok by som chcel popísať niektoré skratky ako sú RDS, TMC a SDR.

- RDS [10] slúži na prenos informácií prostredníctvom FM vysielania. Pomocou RDS napríklad rádio v aute zisťuje názov rádiostanice, či názov skladby ktorá je vysielaná.

- TMC [10] slúži pre poskytovanie aktuálnych dopravných informácií. Jednotlivé dopravné obmedzenia sa kódujú do TMC správ a následne sa prostredníctvom RDS odvysielajú. TMC správy využívajú napríklad navigačné systémy na hľadanie najlepšej trasy.

- SDR [11] je rádiový systém, kde sú hardvérové komponenty, slúžiace napríklad na filtrovanie alebo demoduláciu signálu, nahradené softvérom bežiacim napríklad na počítači.

Celý postup pre získanie TMC JSON tokenov vyzerá tak, že vo firme zachytávame vysielanie z FM vysielania. Pre zachytávanie vysielania používame USB s podporou RTL-SDR a pomocou nástrojov nalaďujeme konkrétnu stanicu. RTL-SDR [11] je USB, ktoré slúži ako počítačový rádiový skener pre príjem rádiových signálov. Následne v Linuxe pomocou nástroja Redsea [12] dekódujeme vysielane z FM do podoby JSON. K programu Redsea máme pripojené lokačné tabuľky, takže môžeme zistiť GPS vysielaných informácií. Lokačné tabuľky sú rovnaké ako sa používajú bežne v autách.

Ja som dostával, už len dekódované správy vo formáte JSON. Aj keď je JSON tokenov v TMC správach viacero typov, do nášho systému sme ich všetky nepotrebovali. Vybrali sme len tých päť, ktoré sme uznali ako najdôležitejšie. Ak by sme ale do budúcna potrebovali aj iné JSON tokeny, nebude problém systém rozšíriť, aby spracovával aj ďalšie potrebné tokeny. Tokeny, ktoré sme sa rozhodli spracovať sú nasledovné.

- Identifikačný token obsahujúci informácie, ktoré určujú, aká stanica dané dáta vysiela. Nachádzajú sa tu údaje ako je názov stanice, alebo frekvencia, na ktorej sa vysielajú dáta. Príklad JSON tokenu je možné vidieť vo výpise 2.

```
{
   "identifier": "test-vltava-sber-37"
   "station": "Radio Vltava",
   "frequency": 104.8,
   "note":"Testovaci RDS-TMC data"
}
```

Výpis 2: Príklad identifikačného tokenu

- Konfiguračný token 1 obsahujúci názov poskytovateľa, ktorý je reprezentovaný kľúčom `service_provider`. Príklad JSON tokenu je možné vidieť vo výpise 3.

```
{
   "group": "8A",
   "pi": "0x232D",
   "prog_type": "Pop music",
   "rx_time": "01/27/20 12:08:11",
   "tmc": {
```

```
      "service_provider": "JSDI-CRo"
   },
   "tp": false
}
```

Výpis 3: Príklad konfiguračného tokenu 1

- Konfiguračný token 2 obsahujúci informácie o lokačnej tabuľke a regionálnom kontexte vysielania. Regionálny kontext je reprezentovaný kľúčom `scope` a jeho hodnotu tvorí pole. V tomto poli sa môže nachádzať buď `national`, `regional`, `international` alebo `urban`. Príklad takéhoto JSON tokenu je možné vidieť vo výpise 4.

```
{
   "group": "3A",
   "open_data_app": {
      "app_name": "RDS-TMC: ALERT-C",
      "oda_group": "8A"
   },
   "pi": "0x232D",
   "prog_type": "Pop music",
   "rx_time": "01/27/20 12:19:17",
   "tmc": {
      "system_info": {
         "is_encrypted": false,
         "is_on_alt_freqs": false,
         "location_table": 25,
         "scope": [
         "national",
         "regional",
         "urban"
         ]
      }
   },
   "tp": false
}
```

Výpis 4: Príklad konfiguračného tokenu 2

- Konfiguračný token 3 obsahujúci informácie o časovaní vysielania dát, ktoré reprezentuje kľúč `gap` a informácie o službe, ktorá je uvedená kľúčom `service_id`. Príklad JSON tokenu je možné vidieť vo výpise 5.

```
{
   "group": "3A",
   "open_data_app": {
      "app_name": "RDS-TMC: ALERT-C",
      "oda_group": "8A"
   },
   "pi": "0x232D",
   "prog_type": "Pop music",
   "rx_time": "01/27/20 12:19:15",
   "tmc": {
      "system_info": {
         "gap": 3,
         "service_id": 3
      }
   },
   "tp": false
}
```

Výpis 5: Príklad konfiguračného tokenu 3

- Token obsahujúci vlastné TMC informácie o dopravnom obmedzení. V tomto tokene sa uvádzajú súradnice, kde sa dané obmedzenie nachádza. Súradníc môže byť aj viacero, ak sa nejedná iba o bod, ale o konkrétny úsek. Ďalej sa tu nachádza informácia pod kľúčom `direction`, označujúca či je dopravné obmedzenie len v jednom smere, alebo v oboch smeroch. Kľúč `description` v JSON tokene nám v hodnote uvádza o aké obmedzenie sa jedná. Kľúče `road_number`, `span_from` a `span_to` nám popisujú názov cesty, na ktorej sa dopravné obmedzenie nachádza. Posledným dôležitým kľúčom je `urgency` popisujúci aké naliehavé je dané obmedzenie. Môže obsahovať buď hodnotu N (normálne), U (urgentné), alebo X (extrémne urgentne). Príklad JSON tokenu je možné vidieť vo výpise 6.

```
{
   "group": "8A",
   "pi": "0x232D",
   "prog_type": "Pop music",
   "rx_time":"01/27/20 12:19:15",
   "tmc": {
      "message": {
         "coordinates": [
         {
            "lat": 48.99902,
```

```
            "lon": 15.35168
          }
          ],
          "description": "Closed due to roadworks.",
          "direction": "both",
          "event_codes": [
          735
          ],
          "extent": "+10",
          "location": 37321,
          "road_number": "II/409",
          "span_from": "Vrat\u011bn\u00edn",
          "span_to": "Horn\u00ed N\u011bm\u010dice",
          "update_class": 5,
          "urgency": "U"
        }
      },
      "tp": false
    }
```


Výpis 6: Príklad tokenu obashujúci vlatné TMC informácie

### 5.3 Modul-1

V module jedna som riešil implementáciu webového modulu pre príjem a filtrovanie TMC tokenov. Najprv som vytvoril triedu v ktorej som implementoval Rest, ktorý bude slúžiť na príjem TMC správ. Pre vytvorenie Restu je potrebné nastaviť cestu pod ktorou sa bude daný rest nachádzať. Pre určenie cesty sa používa anotácia `@Path`. Následne je možné určiť metódy, ktoré chcem používať. Ja som v tomto prípade potreboval, len prijímať správy, tak som naimplementoval funkciu POST. Takže každá správa bude posielaná na tento modul pomocou požiadavky POST na url, ktorá bola definovaná v anotácií `@Path`. Každá takáto správa obsahuje reťazec JSON, ktorý reprezentuje jeden z možných TMC tokenov.

Následne som vytvoril triedu `CheckToken`. V tejto triede som implementoval funkciu, ktorá bude kontrolovať, či daná správa reprezentuje jeden z piatich TMC tokenov, ktoré potrebujeme v našom systéme. Ktoré tokeny využívame v našom systéme som popísal vyššie v sekcií 5.2. Ak správa je pre náš systém nepotrebná tak sa ignoruje. Ak ide o správu, ktorú v našom systéme potrebujeme, tak sa následne posiela daná správa cez streamovaciu službu Kafka na modul-2. V tomto module používam presnejšie Kafka Producer API. Aby som mohol posielať správy cez Kafku vytvoril som triedu `KafkaBean`. V konštruktore danej triedy najprv nastavím potrebné vlastnosti pre Kafka Producer. Následne som implementoval funkciu `sendMessage`, ktorej sa

bude v parametri predávať reťazec, ktorý chceme poslať. Pri posielaní správ, je nutné určiť, na ktorý topic chceme správu poslať.

Takže celý postup vyzerá nasledovne. Správa obsahujúca TMC token je poslaná na Rest. Následne sa správa vyhodnotí, či je potrebná v našom systéme a ak áno posiela sa cez Kafku do modulu-2. Každý JSON token, ktorý prichádza na Rest je v kódovaní UTF-8.

## 5.4 Modul-2

V tejto časti som potreboval nakonfigurovať Kafka Consumer API, aby prijímal JSON správy na topicu `rdstata`. Následne som musel vyriešiť rozloženie JSON správ a ich ukladanie do databáze.

V prvom rade som vytvoril triedu s názvom `KafkaBean`. Tejto triede som pridal potrebné anotácie, aby sa inicializovala automaticky pri štarte projektu. V tejto triede som vytvoril dve funkcie. Prvá funkcia `start` s anotáciou `@PostConstruct`. Daná anotácia môže byť v triede použitá len raz. Tato metóda sa vykoná hneď, ako prvá pri skonštruovaní danej triedy. V tejto metóde som vytvoril inštanciu triedy `KafkaThread`, ktorá je potomkom triedy `Thread`.

V konštruktore triedy `KafkaThread` som vytvoril Kafka Consumer, ktorému som nastavil jej potrebné vlastnosti a vytvoril inštanciu triedy `RdsDataDBWriter`. Po spustení threadu sa v cykly kontroluje, či prišla na topic nová správa. Ak áno a jej hodnota nie je nulová, tak sa správa predá do parametra funkcie `process` triedy `RdsDataDBwriter`.

Keďže viem, že mi môže prísť správa obsahujúca jeden z piatich tokenov, tak najprv zisťujem o aký typ sa jedná. Na rozloženie a prácu s JSON správami používam `JsonParser` a `JsonObject` z balíčka `com.google.gson`. V projekte mám vytvorené triedy, ktoré prezentujú jednotlivé tokeny a zároveň prezentujú atribúty jednotlivých tabuliek z databáze. Vďaka týmto triedam môžem rozložiť každý JSON token priamo do danej triedy.

```
public class TmcSrc {
private Long id;          //id

@SerializedName("identifier")
private String identifier; //identifier

@SerializedName("station")
private String station;   //station

@SerializedName("frequency")
private double frequency; //frequency

@SerializedName("note")
private String note;      //note
```

```
/*
getter/setter
*/

}
```

Výpis 7: Trieda TmcSrc

V triedach je potrebné využívať anotáciu `@SerializedName`, ako je možné vidieť aj na výpise 7, aby program vedel, ktorý dátový typ sa rovná daným kľúčom v objekte JSON. Vo výpise 8 ukazujem funkciu, ktorá vracia vyplnenú inštanciu triedy podľa reťazcov obsahujúci JSON. Funkcia má dva parametre `stringJson` obsahujúci token, ktorý potrebujeme rozložiť a premennú `t`, ktorá je generická. Využil som generickú metódu s generickým parametrom, aby som nemusel vytvarovať pre každú triedu jednu funkciu.

```
public <T> T fillTable(String stringJson, T t) {

Gson gson = new Gson();
JsonParser parser = new JsonParser();
JsonObject object = (JsonObject) parser.parse(stringJson);
t = (T) gson.fromJson(object, t.getClass());
return t;
}
```

Výpis 8: Príklad rozloženia JSON reťazcov na potrebný objekt

Následne ak nejde o inštanciu triedy `TmcSrc` zisťujem, či už predtým prišiel identifikačný token. Ak nám identifikačný token ešte neprišiel, tak sa správa neuloží. Ak nám už predtým identifikačný token prišiel alebo sa jedná o inštanciu triedy `TmcSrc` tak danú inštanciu predávam do jednotlivých funkcii triedy `RdsPostgresJDBC` ako parameter. Dané funkcie mi vracajú int reprezentujúcu primárny kľúč z tabuliek do ktorých boli vložené, aby som ich následne mohol vyplňovať do ďalších záznamoch ako cudzí kľúč podľa potreby.

Trieda `RdsPostgresJDBC` v konštruktore sa najprv zabezpečí, že potrebné tabuľky budú existovať, aké tabuľky sa používajú v našom systéme som popísal v sekcií 5.1 na strane 17. Využívam SQL príkaz `create table if not exist`, čo nám zabezpečí, aby nám nevyvolalo chybu ak by daná tabuľka už existovala. K databáze sa pripájam pomocou JDBC a pre prácu s databázou využívam balíček `java.sql`.

```
public Connection connect() {

Connection connection = null;
```

```
try {
connection = DriverManager.getConnection(url, user, pass);
} catch (Exception e) {
loger.log(Level.WARNING, e.getClass().getName(), e.getMessage());
return null;
}
loger.log(Level.INFO, "Opened database successfully");
return connection;
}
```

Výpis 9: Príklad pripojenia na databázu

V danej funkcií vo výpise 9 môžeme vidieť, že ak sa pripojenie nepodarí vráti nám nulovú hodnotu a vypíše chybu, ak sa ale pripojenie podarí vráti nám objekt typu `Connection`. O pripojenie sa stará metóda `DriverManager.getConnection(url, user, pass)`. Parametre danej funkcie mám ako globálne premenné kde url je reťazec reprezentujúcu url danej databáze. Príklad url: `jdbc:postgresql://localhost:5432/rdsdata`.

## 5.5 Modul-3

V tomto module som riešil vizualizáciu spracovaných dát. Potrebné dáta získavam z databáze, pričom pre každú tabuľku z databáze mám vytvorenú jednu triedu. K databáze sa pripájam rovnakým spôsobom ako som ho popísal v časti modul-2 v sekcií 5.4. Takisto ako v modul-2, aj tu hneď po pripojení vytvorím tabuľky ak neexistujú, aby sme sa vyhýbali zbytočným možným chybám. Vizualizáciu riešim pomocou webu. Na vytvorenie webovej aplikácií používam Wicket framework. Tento modul je najrozsiahlejší. Riešil som tu viacero problémov a naučil sa veľa nových vecí. Preto ho rozdelím do niekoľkých časti. Hlavnou triedou tohto modulu je `RdsWebApplication`, ktorá je potomkom triedy `WebApplication`. Táto trieda je povinnou súčasťou webovej aplikácii, ak chcem používať Wicket. Táto trieda sa automaticky inicializuje pri spustení webu. V konštruktore tejto triedy si môžeme inicializovať všetky potrebné objekty s ktorými budeme chcieť pracovať. Ďalšou dôležitou funkciou v tejto triede je `getHomePage`. Táto funkcia vracia triedu, ktorá je potomkom triedy `Page`. Ide vlastne o triedu, ktorá zobrazuje úvodnú stránku.

### 5.5.1 Nastavenie Wicketu

V tomto projekte musím vytvoriť súbor web.xml v adresáre Web Pages/Web-INF. Tento súbor je automatický vyhľadávaný pri spustení. V tomto súbore musíme nastaviť niekoľko veci.

V prvom rade nastavíme názov triedy, vrátane názvu jeho balíčka, v ktorej máme definovanú hlavnú triedu pre Wicket. Následne určíme pod akým url sa budú Wicketové stránky nachádzať.

Ďalšou, ale nepovinnou informáciou, ktorú tu môžeme nastaviť je dĺžka trvania session. V nasledujúcom výpise 10 je možné vidieť ako takýto súbor vyzerá.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="
    http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns
    .jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
<display-name>RDS collector</display-name>
<filter>
<filter-name>RdsWebApplication</filter-name>
<filter-class>org.apache.wicket.protocol.http.WicketFilter</filter-class>
<init-param>
<param-name>applicationClassName</param-name>
<param-value>cz.eago.rds.air.visualizer.RdsWebApplication</param-value>
</init-param>
</filter>
<filter-mapping>
<filter-name>RdsWebApplication</filter-name>
<url-pattern>/app/*</url-pattern>
</filter-mapping>
<session-config>
<session-timeout>
30
</session-timeout>
</session-config>
</web-app>
```

Výpis 10: Príklad súbora web.xml

### 5.5.2 Všeobecné informácie o webovej aplikácií

Vo webovej aplikácií používam css Bootstrap a šablónu sb-admin-2, aby som zabezpečil lepší vzhľad webových stránok. Bootsrap je voľne šíriteľný a patrí medzi najznámejšie css frameworky. SB-admin-2 je tak isto voľne šíriteľný. A to je dôvod, prečo som sa rozhodol práve pre nich. Celá webová aplikácia sa skladá z dvoch webových stránok. Prvá úvodná stránka s názvom Messages zobrazuje zoznam správ o dopravných obmedzeniach. Druhá stránka s názvom Statiscs je zameraná na zobrazenie štatistík o dopravných obmedzeniach.

### 5.5.3 Stránka Messages

Tato webová stránka, ako som už napísal, zobrazuje dopravné obmedzenia. Dopravné obmedzenia zobrazujem v tabuľke a takisto sa tu nachádza aj mapa, ktorá zobrazuje, kde je dopravné obmedzenie. Celú stránku by som rozdelil na tri časti, ktoré sú filter, tabuľka a mapa.

**5.5.3.1 Tabuľka** Wicket v balíčku obsahuje viacero tried, ktoré nám dokážu pomôcť vytvoriť tabuľku, alebo rôzne zoznamy. Vedel som, že nemôžem zobraziť všetky záznamy pod seba, pretože zoznam bude dosť veľký, ale myslel som si, že keď použijem stránkovanie tak sa problém vyrieši a zobrazím maximálne desať záznamov.

No tu sa vyskytol jeden problém. Síce html stránka zobrazovala len 10 záznamov, ale načítavala všetky, čo spôsobilo, že pri väčšom počte záznamov trvalo dosť dlho, kým sa daná stránka načítala. Preto som musel vymyslieť niečo, aby sa mi načítavali len záznamy, ktoré potrebujem. Zároveň som chcel, aby som naďalej zachoval daný výzor stránky. Nakoniec som sa rozhodol urobiť tabuľku pomocou triedy `RepeatingView`. `RepeatingView` je trieda z balička od Wicketu. Pomocou nej môžem pridávať, iba tie záznamy, ktoré potrebujem.

Záznamy sú vlastné triedy, ktoré sú potomkami triedy `Panel`. Vo Wickete sa stránka skladá z niekoľkých panelov a každý panel môže obsahovať ďalšie panely. Triedy, ktoré sú potomkami triedy `Panel` sa nazývajú komponenty. Cely postup vyzerá následovne.

Najprv z databáze vyberám identifikačné kľúče všetkých záznamov zoradených podľa času vytvorenia. Vďaka tomuto viem presný počet záznamov, ktoré sa v databáze nachádzajú a v akom poradí ich budem chcieť mať na zozname. Následne som vytvoril komponent, ktorý je možné vidieť na výpise 11. V konštruktore tohto komponentu predávam zoznam obsahujúci názvy stĺpcov danej tabuľky a zoznam obsahujúci identifikačné kľúče všetkých záznamov. V tejto komponente som vytvoril ďalšie tri komponenty.

```
public class MyTablePanel extends Panel {
MyTableBodyPanel bodyPanel;
MyTableFooterPanel footerPanel;
MyTableHeaderPanel headerPanel;
int actualPage = 1;
List<Long> ids;
public MyTablePanel(String id, List<String> header, List<Long> ids) {
super(id);
setOutputMarkupId(true);
this.ids = ids;
headerPanel = new MyTableHeaderPanel("tableheader", header);
bodyPanel = new MyTableBodyPanel("tablebody", ids, actualPage);
footerPanel = new MyTableFooterPanel("tablefooter", ids.size(), 10, actual);
add(headerPanel);
```

```
add(bodyPanel);
add(footerPanel);
}
public void setActualPage(int i) {
actualPage = i;
remove(bodyPanel);
bodyPanel = new MyTableBodyPanel("tablebody", ids, actualPage);
add(bodyPanel);
}
```

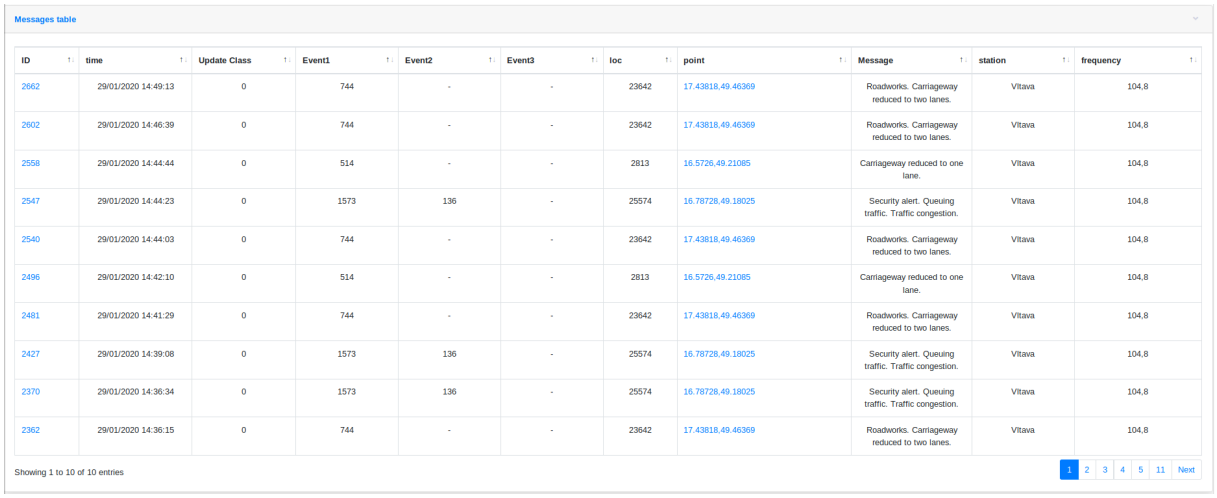
Výpis 11: Trieda MyTablePanel

Prvý komponent reprezentuje hlavičku tabuľky, teda má za úlohu zobrazovať názvy stĺpcov. V konštruktore mu teda v parametri posielam iba zoznam s názvami stĺpcov. Druhý komponent nám zobrazuje telo tabuľky. Táto trieda, ktorú možno vidieť na výpise 12 dostáva v parametri zoznam všetkých identifikačných kľúčov a číslo strany, ktorú má zobrazovať. Rozhodol som sa zobrazovať 10 záznamov na jednu stránku.

```
public class MyTableBodyPanel<T> extends Panel {

WebMarkupContainer cont = new WebMarkupContainer("rows");
RepeatingView listItems = new RepeatingView("dataRow");
List<Long> data;
List<Long> actualData;
int number;
int start;
int lenght;

public MyTableBodyPanel(String id, List<Long> data, int actual) {
super(id);
this.data = data;
/*
* set global data
*/
process();
}
public void process() {
actualData = data.subList(start, start + lenght);
for (Long i : actualData) {
populateitem(i);
}
```




Obr. 2: Grafické znázornenie tabuľky Messages

```
cont.add(listItems);
add(cont);
}
public void populateitem(Long id) {
DataTableItem item = new DataTableItem(listItems.newChildId(), id);
listItems.add(item);
}
}
```

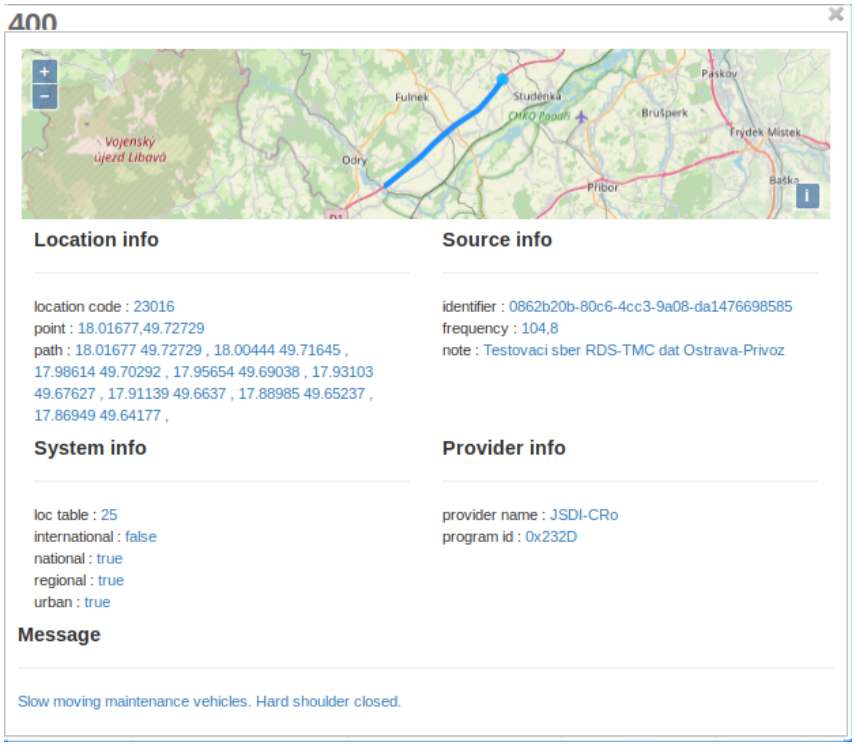
Výpis 12: Trieda MyTableBodyPanel

V tomto komponente, teda viem urobiť nový zoznam desiatich potrebných identifikačných kľúčov z celého zoznamu. Následne v cykly prechádzam daný nový zoznam a pridám pre každý záznam, komponent reprezentujúci riadky tabuľky do objektu typu `RepeatingView`. Každý riadok, čiže komponent vyberie z databáze potrebný záznam podľa identifikačného kľúča, aby mohol vyplniť jednotlivé stĺpce.

Tretí komponent slúži na zobrazenie päty danej tabuľky. V parametri mu predávam celkový počet záznamov a aktuálnu stranu tabuľky. V tomto komponente zobrazujem stránkovanie, teda čísla strán, pomocou ktorých sa môžeme medzi jednotlivými stránkami preklikávať. Vždy zobrazujem možnosť výberu prvej a poslednej stránky a päť najbližších stránok k aktuálnej stránke. Po kliknutí na číslo požadovanej stránky sa prepíše aj celý komponent zobrazujúci telo tabuľky.

Tieto komponenty mi, teda zaručili bezproblémové zobrazovanie tabuľky a aby načítavanie stránok bolo dostatočne rýchle. Vzhľad tabuľky som doriešil už priamo v html kóde.

V tabuľke sú stĺpce zobrazujúce informácie, ako je možné vidieť aj na obrázku 2, ktoré sme po dohode s vedúcim v práci uznali, že sú tie najdôležitejšie. Je možné zobraziť aj podrobnejšie




Obr. 3: Grafické znázornenie modálneho okna

informácie ku každému záznamu. Podrobnejšie informácie sa zobrazia v modálnom okne po kliknutí na id záznamu.

Na vytvorenie modálneho okna používam triedu `ModalWindow`, ktorá je obsahom balička Wicket. Modálnemu oknu je potrebné predať do funkcie `setPageCreator` ako parameter stránku, ktorá sa ma v ňom zobraziť. Ďalšou, podľa mňa dôležitou informáciou, je možnosť nastaviť, čo sa ma po zatvorení okna vykonať. Napríklad či sa ma obnoviť stránka, alebo niečo iné. Táto možnosť sa nastavuje pomocou metódy `setWindowClosedCallback`. V tomto prípade som ja nepotreboval nič konkrétne vykonať tak som nechal danú funkciu prázdnu.

Pre modálne okno ktoré je možné vidieť na obrázku 3, mám vytvorenú vlastnú triedu s názvom `ItemDetail`. Tato trieda ma ako parameter v konštruktore id záznamu, pomocou ktorého získa z databáze všetky potrebné detaily. V tomto okne zobrazujem aj mapu s vyznačením, kde sa dopravné obmedzenie nachádza.

Ak chceme vidieť na mape, kde sa dane obmedzenie nadchádza, bez toho, aby sme si museli otvárať modálne okno stačí kliknúť na súradnice v danom zázname. Automaticky nás presunie nižšie ku mape, kde nám následne zobrazí a priblíži bod, kde sa dané dopravné obmedzenie nachádza.

**5.5.3.2 Mapa** Na vytvorenie mapy som potreboval pridať novú knižnicu od wicketstuff konkrétne wicketstuff-openlayers3, ktorá dopĺňa klasicky Wicket o potrebné triedy na vytváranie mapy. Táto knižnica funguje tak, že triedy obsahujú reťazce reprezentujúce kód z JavaScriptu.

Pomocou nastavení daných tried sa vytvorí reťazec, ktorý sa následne preloží na JavaScriptový kód.

Konkrétne som využil triedu `DefaultOpenLayersMap`, ktorá nám vykresli mapu, ale treba jej v konštruktore predať model mapy. Model mapy sa skladá z dvoch objektov.

Prvý takýto objekt je `View`. `View` nám určí východzie vycentrovanie mapy na určitý bod a nastaví nám zoom. Tieto dve vlastnosti mu predávame v konštruktore. Bod, na ktorý chceme mapu vycentrovať uvádzame pomocou objektu `LongLat`, v ktorom zadáme zemepisnú šírku a výšku. Zoom je typu `Integer` a ja som ho nastavil na hodnotu deväť.

Druhý objekt je zoznam typu `Layer`. Do tohto zoznamu pridávame dáta, ktoré potrebujeme zobraziť v mape.

Najprv musíme do zoznamu pridať objekt `Tile`, ktorý je potomkom triedy `Layer`. V objekte `Tile` musíme určiť zdroj mapy. Ja používam konkrétne OpenStreetMap. Na výber ich máme ale viacero.

Následne potrebujem do zoznamu pridať body a úsečky, ktoré budú zobrazovať, kde je dopravné obmedzenie. Pre pridanie bodov, alebo úsečiek sa používa trieda `Vector`, ktorá je tiež potomkom triedy `Layer`. Triede `Vector` musíme v konštruktore pridať objekt `VectorSource`, do ktorého vložíme zoznam bodov, ktoré chcem zobraziť. Ja som vytvoril dve inštancie triedy `Vector`, jednu pre zobrazenie bodov a druhú pre zobrazenie úsečiek, čiže som vytvoril dve inštancie `VectorSource` a samozrejme aj dva zoznamy typu `Layer`.

V cykle som následne prechádzal záznamy z tabuliek, z ktorých som získaval body a pridával ich do zoznamu. Ak záznam obsahoval nie len bod, ale aj cestu tak som tuto cestu pridával do druhého zoznamu.

Na vytvorenie bodu existuje objekt `Point`, ktorému stačí zadať súradnice. Následne tento bod stačí predať ako parameter objektu `PersistentFeature`, ktorý je potomkom objektu `Featuru` a určiť, ako ma daný bod vyzerať pomocou objektu `Style`. Nastavil som mu len farbu a veľkosť.

S vytvorením úsečky to už, ale nebolo tak ľahké. Keďže som nenašiel vhodný objekt, ktorý by mi potrebnú cestu vytvoril musel som si vytvoriť vlastnú triedu, ktorú som pomenoval `LineGeometry`. Táto trieda dedí z triedy `PersistentFeature` a ako parameter v konštruktore jej predávam pole objektov typu `Point`, obsahujúci súradnice cesty. Prvom rade som si preštudoval, ako sa dá vytvoriť čiara, alebo úsečka pomocou JavaScriptu. Po preštudovaní som potreboval prepísať dve zdedené funkcie a to `renderAfterConstructorJs` a `renderJs`. Funkcia `renderJs` vracia reťazec obsahujúca JavaScriptový kód na vytvorenie čiary.

```
 @Override
public String renderJs() {
   StringBuilder builder = new StringBuilder();
   builder.append("{");
      builder.append("'id': \"" + getJsId() + "\",");

      if (getGeometry() != null) {
```

```
            builder.append("'geometry': new ");
            builder.append("ol.geom.LineString");
            builder.append("([");
            boolean first = false;
            for (Point i : geometrys) {
               if (first) {
                  builder.append(",");
               }
               first = true;
               builder.append(i.renderJs());
            }
            builder.append("]),");
         }

         if (getName() != null) {
            builder.append("'name': '" + Strings.escapeMarkup(getName()).toString()
                 + "',");
         }

         builder.append("}");
      return builder.toString();
   }
```

Výpis 13: Funkcia rednerJS

Vo výpise 13 sa vytvára objekt, konkrétne objekt typu `ol.geom.LineString`, ktorému sa v konštruktore predáva ako parameter pole bodov. Aby som dane pole bodov mohol vložiť do reťazca, tak som ho musel v cykle prechádzať. Body typu Point taktiež obsahujú funkciu `renderJs`, ktorá vracia reťazec a tak som jednotlivé reťazce pridával do nášho jedného potrebného reťazca.

Funkcia `renderAfterConstructorJs` vracia tak isto reťazec obsahujúci JavaScriptový kód, ktorý slúži na určenie štýlu, alebo presnejšie povedané na určenie, ako má daná čiara vyzerať.

```
    @Override
public String renderAfterConstructorJs() {
StringBuilder builder = new StringBuilder();

builder.append(getJsId());
builder.append(".setStyle(new ");
builder.append("ol.style.Style({");
builder.append(" stroke: new ol.style.Stroke({");
```

```
builder.append("color: '#1E90FF',");
builder.append("width: 5");
builder.append("})}");
builder.append("));");

return builder.toString();
}
```

Výpis 14: Funkcia renderAfterConstructorJs

Ako je vidieť na výpise 14 táto funkcia bola jednoduchšia. Stačilo vytvoriť štýl a nastaviť potrebnú farbu čiare a šírku danej čiary.

A to je asi všetko potrebné na vytvorenie čiary takže mapu máme hotovú. Len tu nastal tak isto podobný problém, ako pri tabuľke, že sa všetky body zbytočne načítavali naraz a vo výsledku to tak isto spomaľovalo načítanie stránky. Rozhodol som sa preto načítavať, len tie body, ktoré budú vidno na mape. Teda po presúvaní mapy sa mi jednotlivé body budú načítavať. Posúvať zobrazenie mapy sa dá klasicky pomocou stlačenia ľavého tlačítka myši a následne s pohybom myši si môžeme prezerať mapu.

Našiel som objekt `ViewEventListner` obsahujúci funkciu, ktorá sa vyvolala vždy, ak sa pohlo s mapou. Stačilo ju len k mape priradiť, ale pri každom pohybe sa daná funkcia vyvolala príliš veľa krát, keďže reagovala aj na veľmi malé pohyby, čo mi spôsobovalo problémy. Preto som sa snažil nájsť objekt s funkciou, ktorá sa vyvolá až, keď sa pohyb na mape skonči. Čiže nie neustále pri pohybe, ale len raz a až na konci pohybu. Ale takýto objekt s potrebnou funkciou som nenašiel, preto som si musel vytvoriť vlastnú triedu.

Hľadal som spôsob, ako by sa dala urobiť trieda s potrebnou funkcionalitou. Rozhodol som sa najprv teda preštudovať, ako funguje trieda `ViewEventListner` a podľa nej urobiť vlastnú triedu. Zistil som, že na vytvorenie danej triedy je treba vytvoriť aj súbor typu JavaScript, ktorý daná trieda využíva.

`ViewEventListner` je abstraktná trieda a obsahuje abstraktnú funkciu `handleViewEvent` s parametrami `target` typu `AjaxRequestTarget` a `viewEvent` typu `ViewEvent`. Ďalej obsahuje funkciu `respond`, ktorá je vyvolaná ako odpoveď na požiadavku z JavaScriptu. Pri jej vyvolaní sa vypĺňajú aj parametre. Celkovo to funguje ako požiadavka na server, či už typu GET, alebo POST, ktorá ma vyplniť určitý potrebný parameter. Pri vyvolaní tejto funkcií sa vypĺňuje reťazec obsahujúci JSON. Tento JSON sa následne rozloží a vytvorí sa z neho objekt typu `ViewEvent`. Nakoniec sa vyvolá abstraktná funkcia `handleViewEvent`, ktorej sa v parametri daný objekt predáva.

Posledná funkcia v tejto triede je `renderHead`. V tejto funkcií sa zadefinujú parametre pomocou objektu `Map`. Následne sa preloží JavaScriptový súbor ViewEventListner.js s danými parametrami. Tuto triedu som nakoniec vôbec nemusel meniť. Potreboval som zmeniť len JavaScriptový súbor.

V tomto JavaScriptovom súbore sa nachádza funkcia, ktorú je možne vidieť na výpise 15 a jej názov sa odvádza od parametra callbackFunctionName.

```
${callbackFunctionName} = function(view)
{
   Wicket.Ajax.post(
   {'u': '${callbackUrl}',
      'dep': [function() {
         return {'view': view,};
      }]});
};
```

Výpis 15: Funkcia callbackFunctionName

Pomocou tejto funkcie sa vyvolá funkcia respond z triedy ViewEventListner s parametrom view. Tuto funkciu som tak isto ponechal.

Ďalšia funkcia, ktorá sa v tomto súbore nachádza a ktorú som aj potreboval zmeniť je tá, ktorá reagovala na zmeny. Táto funkcia vyvolávala pri každom pohybe mapy funkciu callbackFunctionName a ja som ju potreboval zmeniť, aby sa vyvolávala až po ukončení pohybu na mape. Toto som urobil tak, že som nastavil vyvolávanie na moveend.

```
window.org_wicketstuff_openlayers3['map_${componentId}'].on('moveend', function
    (event)
{
   var view=null;
   view = event.map.getView().getProperties();
   if('${projection}' != 'NULL')
   {
      extent = ol.extent.applyTransform(window.org_wicketstuff_openlayers3['
          map_${componentId}'].getView().calculateExtent(window.
          org_wicketstuff_openlayers3['map_${componentId}'].getSize()),
      ol.proj.getTransform(window.org_wicketstuff_openlayers3['map_${
          componentId}'].getView().getProjection(),
      '${projection}'));
      view['transformedExtent'] = extent;
      view['transformedProjection'] = '${projection}';
   }
   ${callbackFunctionName}(JSON.stringify(view));
});
```

Výpis 16: Funkcia reagujúca na pohyb mapy

Vo výpise 16 sa najskôr vytvorí premenná `view`, ktorá slúži pre získanie aktuálnych nastavení. Následne sa do pramennej `extent` vloží aktuálne zobrazenie, ktoré sa nakoniec pridá do `view`. Pomocou tohto dokážeme získať, kde sme sa pomocou mapy presunuli. Z objektu `view` sa následne vytvorí reťazec obsahujúci JSON, ktorý sa následne predá funkcií `callbackFunctionName` ako parameter.

Takúto triedu som následne pridal do mapy a implementoval som abstraktnú funkciu `handleViewEvent` . V tejto funkcií zistím aktuálne zobrazenie mapy a podľa neho určím, ktoré body sa majú vykresliť na mape.

**5.5.3.3 Filter** Filter nám umožňuje filtrovať záznamy, či už v tabuľke, alebo v mape, a to buď podľa času, podľa update class, podľa event code1, podľa location code a podľa identiefier. Vzhľad filtra môžeme vidieť na obrázku 4.

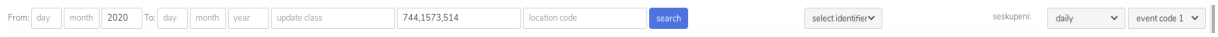
Filtrovať podľa update class, event code1 a location code sa dá pomocou zoznamu oddeleného čiarkou.

Pomocou času môžeme filtrovať odvysielané správy v definovanom období. Čiže môžeme vyfiltrovať od ktorého dňa, po ktorý deň chceme zobrazovať záznamy. Filtrovanie je vo formante od dd/mm/rrrr do dd/mm/rrrr pričom každý ma svoj vlastný input.

- Ak filter podľa času nie je vyplnený, zobrazujú sa iba záznamy za dnes.
- Ak je zadaný iba deň od kedy( OD dd), filter platí pre zadaný deň aktuálneho mesiaca a roka.
- AK je zadaný iba deň a mesiac od kedy(OD dd/mm), filter platí pre daný deň v zadanom mesiaci aktuálneho roka.
- AK je zadaný iba deň, mesiac a rok od kedy(OD dd/mm/rrrr), filter platí pre daný deň v zadanom mesiaci a roku.
- Ak je zadaný iba mesiac od kedy (OD mm), filter platí pre celý zadaný mesiac aktuálneho roka.
- Ak je zadaný iba mesiac a rok od kedy(OD mm/rrrr),filter platí pre celý zadaný mesiac v danom roku.
- Ak je zadaný iba rok od kedy(OD rrrr),filter platí pre celý zadaný rok.
- V ostatných prípadoch je potrebné kompletne zadať obdobie, pre ktoré ma filter platiť, ak ale nebude zadaný rok, filter bude rátať s aktuálnym rokom.

Filtrovanie podľa identifier je možné podľa select boxu, kde je zoznam všetkých dostupných kľúčov identifier. Po kliknutí na tlačítko search sa nastaví filter a aby som vedel prenastaviť tabuľku a mapu na nové hodnoty filtru, vytvoril som pomocnú triedu `Utils`.


Obr. 4: Grafické znázornenie filtru na stránke Messages



Obr. 5: Grafické znázornenie filtru na stránke Statistics

V tejto triede som vytvoril interface `FilterDelegate` s abstraktnou funkciou `invoke`. Ďalej som si zadefinoval objekt `FilterEvent` typu `Map`, kde kľúč je reprezentovaný reťazcom a hodnota je typu `FilterDelegate`. A ako posledne som v tejto triede vytvoril funkciu `RaiseFilterEvent`. V tejto funkcií prechádzam v cykle jednotlivé hodnoty z objektu `FilerEvent` a pre každú hodnotu zavolám funkciu `invoke`. Následne som vytvoril inštanciu tejto triedy v triede `RdsWebApplication` a zadefinoval som funkciu `getWebUtils`, ktorá mi vracia danú inštanciu. S týmto si dokážem zabezpečiť prístup k danej inštancií z každej triedy v projekte.

Nakoniec som si v potrebných triedach pridal do objektu `FilerEvent` z triedy `Utils` nový záznam. Kľúč reprezentoval názov triedy a ako hodnotu som priradil interface s implementovanou funkciou, v ktorej som vykonal potrebné inštrukcie na prekreslenie mapy, či prepísanie tabuľky. Takže po stlačení tlačítka search sa zavolá funkcia `RaiseFilterEvent` a v každej triede, ktorá bola pridaná do zoznamu sa zavolá implementovaná funkcia.

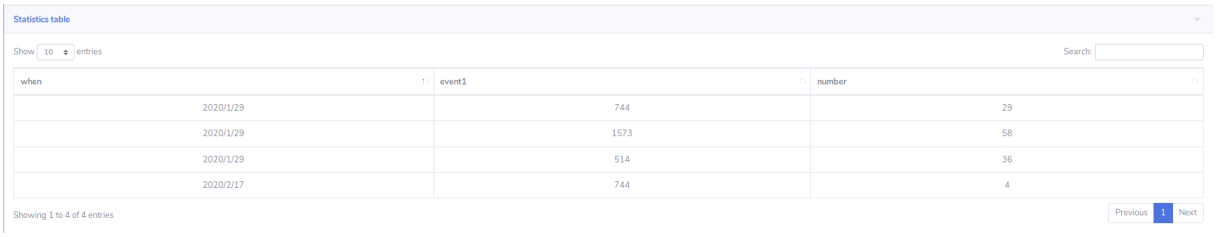
### 5.5.4 Stránka Statistics

Na tejto stránke, ako som už písal sa nachádza taktiež filter, tabuľka a graf so štatistikou.

**5.5.4.1 Filter** Filter v tejto stránke je rovnaký ako aj na stránke Messages, ktorú som popísal v kapitole 5.5.3 na strane 27, ale je rozšírený o zoskupovanie, ako je možné vidieť aj na obrázku 5.Vždy po výbere novej hodnoty sa automaticky prepíše ako tabuľka tak aj graf. Zoskupovanie je možné podľa času a zároveň aj podľa typu.

- Pre zoskupovanie podľa času máme na výber z troch možnosti buď denne, mesačne alebo môžeme vybrať možnosť za celé obdobie. Ako východzia hodnota je nastavená možnosť za celé obdobie.
- Pre zoskupovanie podľa typu, máme na výber zoskupovať buď podľa update class, event code1, alebo location code. Ak nechceme zoskupovať podľa typu, môžeme vybrať možnosť (by type), ktorá je aj východzia hodnota.

**5.5.4.2 Tabuľka** Tabuľka sa skladá z troch stĺpcov when, type a number. Názov stĺpca type sa mení na základe aktuálneho zoskupenia podľa typu. Ak, teda vyberieme zoskupovanie podľa location code, názov stĺpca sa premenuje na location code. V tomto stĺpci sa zobrazujú hodnoty




Obr. 6: Grafické znázornenie tabuľky obsahujúcej štatistiky

podľa vybraného zoskupenia. Ak nezoskupujeme podľa typu, teda máme vybratú možnosť (by type), tak sa v tomto stĺpci vypíše all;

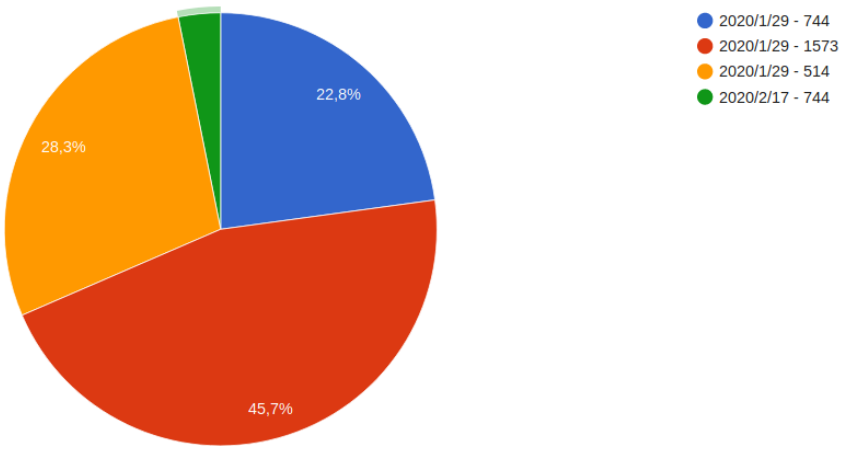
Stĺpec when nám vypisuje buď Whole period ak máme vybranú možnosť zoskupenia za celé obdobie, alebo dátum odpovedajúci danému záznamu. Ak máme vybrané zoskupovanie podľa dňa zobrazuje sa dátum vo formáte rrrr/mm/dd, ak ale máme zoskupovanie podľa mesiaca zobrazí sa dátum len vo formáte rrrr/mm. Posledný stĺpec number nám zobrazuje počet dopravných obmedzení podľa aktuálneho zoskupenia. Celé rozloženie tabuľky je možné vidieť na obrázku 6.

**5.5.4.3 Graf** Graf nám rovnako zobrazuje štatistiky podľa aktuálneho zoskupenia. Rozhodol som sa použiť kruhový graf. Na vytvorenie grafu som potreboval pridať novú knižnicu, ktorá je tak isto od wicketstuff konkrétne wicket-gchart. Z tejto knižnice som použil triedu `Chart`, ktorá zabezpečuje vytvorenie potrebného grafu. Funguje to podobne ako v prípade vytvárania mapy. Najprv sa vytvoria triedy, ktoré sa nastavia podľa potrebných vlastností a následne sa preložia do JavaScriptového formátu. Triede `Chart` sa v konštruktore predávajú tri modely.

Pomocou prvého modelu určujeme typ grafu. Na výber je viacero typov grafov, či už stĺpcový, čiarový, alebo kruhový tiež označovaný ako koláčový. Ja som sa rozhodol, ako som už písal pre kruhový. Pomocou druhého modelu môžeme určiť rôzne nastavenia pre daný graf. Napríklad určiť výšku, či šírku daného grafu. Ja som v tomto modelu nič presne nenastavoval, potrebnú výšku a šírku som si určil priamo v html kóde.

Najdôležitejší je, ale tretí model. Ten je zložený z dvoch zoznamov. Jeden zoznam je pre definovanie typu hodnôt, ktoré do daného grafu vkladáme. Tu je tiež možnosť viacerých typov. Ja pridávam do zoznamu potrebné dva typy. Jeden typ je reprezentovaný reťazcom, ktorý bude určovať názov jednotlivých hodnôt, ktoré budu v grafe. A druhý typ ktorý som pridal je `Number`. Tento typ bude prezentovať hodnotu podľa aktuálneho zoskupenia.

Druhý zoznam reprezentuje hodnoty, ktoré do grafu pridávame. Každý záznam, ktorý pridávame do tohto zoznamu sa skladá z dvoch časti. Prvá časť ja názov a druhá počet. Ich typy zodpovedajú typom, ktoré som zadefinoval v predchádzajúcom zozname. Názov reprezentuje spojenie prvých dvoch stĺpcov when a type z predchádzajúcej tabuľky v sekcií 5.5.4.2. Počet reprezentuje stĺpec number z danej tabuľky. Na obrázku 7 je možné vidieť ako celý graf vyzerá.




Obr. 7: Ukážkový graf

# 6 Záver

V dokumente som popísal informácie o firme a akým projektom sa venuje, technológie, ktoré som použil pri danom projekte, a tak isto som popísal jednotlivé kroky, ako som riešil vývoj a s akými problémami som sa stretol.

Počas praxe som sa naučil používať viacero nových technológií, či už ide o Wicket, alebo o streamovaciu službu Kafka. Narazil som síce na viaceré problémy, ale podarilo sa mi ich po preštudovaní vyriešiť a dotiahnuť projekt do úspešného konca. Vytvoril som celý aplikačný systém na monitorovanie dopravných obmedzení, či už ich získavanie, alebo zobrazovanie na webovej stránke.

V projekte by sa dalo v budúcnosti vylepšiť ešte pár veci, poprípade ho rozšíriť o nové funkcionality. Ako je napríklad urobiť ďalší modul, ktorý bude pristupovať do databáze, a tak by sa nemusel implementovať databázový prístup v dvoch moduloch. Zároveň by to mohlo zlepšiť aj bezpečnosť.

Celá odborná bakalárska prax mi dala možnosť zistiť, ako funguje vývoj systémov vo firme. Či už ide o návrh, implementáciu, alebo testovanie. Zistil som, že pri vývoji každej aplikácií sa vyskytnú chyby, alebo problémy, ktoré je potrebné odstrániť. Zároveň je potrebné pri každej aplikácii naštudovať nové technológie a snažiť sa optimalizovať aplikáciu, čo najlepšie.

Počas celej doby mi vedenie firmy vychádzalo v ústrety, či už som potreboval voľno kvôli škole, alebo kvôli iným záležitostiam. Keď som si niečím nebol istý mi bez problémov pomohli a s radosťou prekonzultovali každý môj problém, či moju pripomienku. V danej firme viem, že budem ešte dlho pokračovať a zúčastňovať sa na ďalších projektov, či už väčších spoločných, alebo menších samostatných.

Myslím si, že výber odbornej praxe je dobrá možnosť, nakoľko si študent môže spojiť teóriou s praxou. Poprípade sa naučiť nové technológie. Zároveň každý študent môže zistiť, ako funguje vývoj vo firme. Tak isto si uvedomí, že teoretické znalosti, ktoré sa naučí v škole, sú dôležité na to, aby sa mohol v budúcnosti podieľať na vývoji aplikácií v praxi. A hlavne študent zisti, že v technickom odvetí sa musí neustále vzdelávať, aby bol schopný používať najnovšie technológie.

# Literatúra


1. *Eago systems spol. s.r.o.* [online] [cit. 2020-04-14]. Dostupné z: https://www.eago.cz/.
2. *Java* [online] [cit. 2020-04-14]. Dostupné z: https://openjdk.java.net/.
3. *Netbeans* [online] [cit. 2020-04-14]. Dostupné z: https://www.oracle.com/tools/technologies/netbeans-ide.html.
4. *Maven* [online] [cit. 2020-04-14]. Dostupné z: https://maven.apache.org/.
5. *PostgreSQL* [online] [cit. 2020-04-14]. Dostupné z: https://www.postgresql.org/.
6. *Wildfly* [online] [cit. 2020-04-14]. Dostupné z: https://wildfly.org/.
7. *Apache Wicket* [online] [cit. 2020-04-14]. Dostupné z: https://wicket.apache.org/.
8. *Apache Kafka* [online] [cit. 2020-04-14]. Dostupné z: https://kafka.apache.org/.
9. *Rest* [online] [cit. 2020-04-14]. Dostupné z: http://wiki.cs.vsb.cz/images/f/fd/TAMZ-cv-11-CZ.pdf.
10. *RDS TMC* [online] [cit. 2020-04-14]. Dostupné z: https://auto.sme.sk/c/2495917/co-je-to-rds-tmc.html.
11. *RTL-SDR* [online] [cit. 2020-04-14]. Dostupné z: https://www.rtl-sdr.com/about-rtl-sdr/.
12. *Redsea* [online] [cit. 2020-04-14]. Dostupné z: https://github.com/windytan/redsea.